Process Models in Software Engineering

Walt Scacchi, Institute for Software Research, University of California, Irvine

February 2001

Revised Version, May 2001, October 2001

Final Version to appear in, J.J. Marciniak (ed.), *Encyclopedia of Software Engineering, 2nd Edition*, John Wiley and Sons, Inc, New York, December 2001.

Introduction

Software systems come and go through a series of passages that account for their inception, initial development, productive operation, upkeep, and retirement from one generation to another. This article categorizes and examines a number of methods for describing or modeling how software systems are developed. It begins with background and definitions of traditional software life cycle models that dominate most textbook discussions and current software development practices. This is followed by a more comprehensive review of the alternative models of software evolution that are of current use as the basis for organizing software engineering projects and technologies.

Background

Explicit models of software evolution date back to the earliest projects developing large software systems in the 1950's and 1960's (Hosier 1961, Royce 1970). Overall, the apparent purpose of these early software life cycle models was to provide a conceptual scheme for rationally managing the development of software systems. Such a scheme could therefore serve as a basis for planning, organizing, staffing, coordinating, budgeting, and directing software development activities.

Since the 1960's, many descriptions of the classic software life cycle have appeared (e.g., Hosier 1961, Royce 1970, Boehm 1976, Distaso 1980, Scacchi 1984, Somerville 1999). Royce (1970) originated the formulation of the software life cycle using the now familiar "waterfall" chart, displayed in Figure 1. The chart summarizes in a single display how developing large software systems is difficult because it involves complex engineering tasks that may require iteration and rework before completion. These charts are often employed during introductory presentations, for people (e.g., customers of custom software) who may be unfamiliar with the various technical problems and strategies that must be addressed when constructing large software systems (Royce 1970).

These classic software life cycle models usually include some version or subset of the following activities:

- *System Initiation/Planning:* where do systems come from? In most situations, new

feasible systems replace or supplement existing information processing mechanisms whether they were previously automated, manual, or informal.

- *Requirement Analysis and Specification:* identifies the problems a new software system is suppose to solve, its operational capabilities, its desired performance characteristics, and the resource infrastructure needed to support system operation and maintenance.

- *Functional Specification or Prototyping*: identifies and potentially formalizes the objects of computation, their attributes and relationships, the operations that transform these objects, the constraints that restrict system behavior, and so forth.

- *Partition and Selection* (Build vs. Buy vs. Reuse): given requirements and functional specifications, divide the system into manageable pieces that denote logical subsystems, then determine whether new, existing, or reusable software systems correspond to the needed pieces.

- *Architectural Design and Configuration Specification*: defines the interconnection and resource interfaces between system subsystems, components, and modules in ways suitable for their detailed design and overall configuration management.

- *Detailed Component Design Specification*: defines the procedural methods through which the data resources within the modules of a component are transformed from required inputs into provided outputs.

- *Component Implementation and Debugging*: codifies the preceding specifications into operational source code implementations and validates their basic operation.

- *Software Integration and Testing*: affirms and sustains the overall integrity of the software system architectural configuration through verifying the consistency and completeness of implemented modules, verifying the resource interfaces and interconnections against their specifications, and validating the performance of the system and subsystems against their requirements.

- *Documentation Revision and System Delivery*: packaging and rationalizing recorded system development descriptions into systematic documents and user guides, all in a form suitable for dissemination and system support.

- *Deployment and Installation*: providing directions for installing the delivered software into the local computing environment, configuring operating systems parameters and user access privileges, and running diagnostic test cases to assure the viability of basic system operation.

- *Training and Us*e: providing system users with instructional aids and guidance for understanding the system's capabilities and limits in order to effectively use the system.

- *Software Maintenance*: sustaining the useful operation of a system in its host/target environment by providing requested functional enhancements, repairs, performance

improvements, and conversions.

What is a software life cycle model?

A software life cycle model is either a descriptive or prescriptive characterization of how software is or should be developed. A descriptive model describes the history of how a particular software system was developed. Descriptive models may be used as the basis for understanding and improving software development processes, or for building empirically grounded prescriptive models (Curtis, Krasner, Iscoe, 1988). A prescriptive model prescribes how a new software system should be developed. Prescriptive models are used as guidelines or frameworks to organize and structure how software development activities should be performed, and in what order. Typically, it is easier and more common to articulate a prescriptive life cycle model for how software systems should be developed. This is possible since most such models are intuitive or well reasoned. This means that many idiosyncratic details that describe how a software systems is built in practice can be ignored, generalized, or deferred for later consideration. This, of course, should raise concern for the relative validity and robustness of such life cycle models when developing different kinds of application systems, in different kinds of development settings, using different programming languages, with differentially skilled staff, etc. However, prescriptive models are also used to package the development tasks and techniques for using a given set of software engineering tools or environment during a development project.

Descriptive life cycle models, on the other hand, characterize how particular software systems are actually developed in specific settings. As such, they are less common and more difficult to articulate for an obvious reason: one must observe or collect data throughout the life cycle of a software system, a period of elapsed time often measured in years. Also, descriptive models are specific to the systems observed and only generalizable through systematic comparative analysis. Therefore, this suggests the prescriptive software life cycle models will dominate attention until a sufficient base of observational data is available to articulate empirically grounded descriptive life cycle models.

These two characterizations suggest that there are a variety of purposes for articulating software life cycle models. These characterizations serve as a

- Guideline to organize, plan, staff, budget, schedule and manage software project work over organizational time, space, and computing environments.
- Prescriptive outline for what documents to produce for delivery to client.
- Basis for determining what software engineering tools and methodologies will be most appropriate to support different life cycle activities.
- Framework for analyzing or estimating patterns of resource allocation and consumption during the software life cycle (Boehm 1981)
- Basis for conducting empirical studies to determine what affects software productivity, cost, and overall quality.

What is a software process model?

In contrast to software life cycle models, software process models often represent a networked sequence of activities, objects, transformations, and events that embody strategies for accomplishing software evolution. Such models can be used to develop more precise and formalized descriptions of software life cycle activities. Their power emerges from their utilization of a sufficiently rich notation, syntax, or semantics, often suitable for computational processing.

Software process networks can be viewed as representing multiple interconnected task chains (Kling 1982, Garg 1989). Task chains represent a non-linear sequence of actions that structure and transform available computational objects (resources) into intermediate or finished products. Non-linearity implies that the sequence of actions may be non-deterministic, iterative, accommodate multiple/parallel alternatives, as well as partially ordered to account for incremental progress. Task actions in turn can be viewed a non-linear sequences of primitive actions which denote atomic units of computing work, such as a user's selection of a command or menu entry using a mouse or keyboard. Winograd and others have referred to these units of cooperative work between people and computers as "structured discourses of work" (Winograd 1986), while task chains have become popularized under the name of "workflow" (Bolcer 1998).

Task chains can be employed to characterize either prescriptive or descriptive action sequences. Prescriptive task chains are idealized plans of what actions should be accomplished, and in what order. For example, a task chain for the activity of object-oriented software design might include the following task actions:

- Develop an informal narrative specification of the system.
- Identify the objects and their attributes.
- Identify the operations on the objects.
- Identify the interfaces between objects, attributes, or operations.
- Implement the operations.

Clearly, this sequence of actions could entail multiple iterations and non-procedural primitive action invocations in the course of incrementally progressing toward an object-oriented software design.

Task chains join or split into other task chains resulting in an overall production network or web (Kling 1982). The production web represents the "organizational production system" that transforms raw computational, cognitive, and other organizational resources into assembled, integrated and usable software systems. The production lattice therefore structures how a software system is developed, used, and maintained. However, prescriptive task chains and actions cannot be formally guaranteed to anticipate all possible circumstances or idiosyncratic

foul-ups that can emerge in the real world of software development (Bendifallah 1989, Mi 1990). Thus, any software production web will in some way realize only an approximate or incomplete description of software development.

Articulation work is a kind of unanticipated task that is performed when a planned task chain is inadequate or breaks down. It is work that represents an open-ended non-deterministic sequence of actions taken to restore progress on the disarticulated task chain, or else to shift the flow of productive work onto some other task chain (Bendifallah 1987, Grinter 1996, Mi 1990, Mi 1996, Scacchi and Mi 1997). Thus, descriptive task chains are employed to characterize the observed course of events and situations that emerge when people try to follow a planned task sequence. Articulation work in the context of software evolution includes actions people take that entail either their accommodation to the contingent or anomalous behavior of a software system, or negotiation with others who may be able to affect a system modification or otherwise alter current circumstances (Bendifallah 1987, Grinter 1996, Mi 1990, Mi 1996, Scacchi and Mi 1997). This notion of articulation work has also been referred to as software process dynamism.

Traditional Software Life Cycle Models

Traditional models of software evolution have been with us since the earliest days of software engineering. In this section, we identify four. The classic software life cycle (or "waterfall chart") and stepwise refinement models are widely instantiated in just about all books on modern programming practices and software engineering. The incremental release model is closely related to industrial practices where it most often occurs. Military standards based models have also reified certain forms of the classic life cycle model into required practice for government contractors. Each of these four models uses coarse-grain or macroscopic characterizations when describing software evolution. The progressive steps of software evolution are often described as stages, such as requirements specification, preliminary design, and implementation; these usually have little or no further characterization other than a list of attributes that the product of such a stage should possess. Further, these models are independent of any organizational development setting, choice of programming language, software application domain, etc. In short, the traditional models are context-free rather than context-sensitive. But as all of these life cycle models have been in use for some time, we refer to them as the traditional models, and characterize each in turn.

Classic Software Life Cycle

The classic software life cycle is often represented as a simple prescriptive waterfall software phase model, where software evolution proceeds through an orderly sequence of transitions from one phase to the next in order (Royce 1970). Such models resemble finite state machine descriptions of software evolution. However, these models have been perhaps most useful in helping to structure, staff, and manage large software development projects in complex organizational settings, which was one of the primary purposes (Royce 1970, Boehm 1976). Alternatively, these classic models have been widely characterized as both poor descriptive and prescriptive models of how software development "in-the-small" or "in-the-large" can or should occur. Figure 1 provides a common view of the waterfall model for software development attributed to Royce (1970).

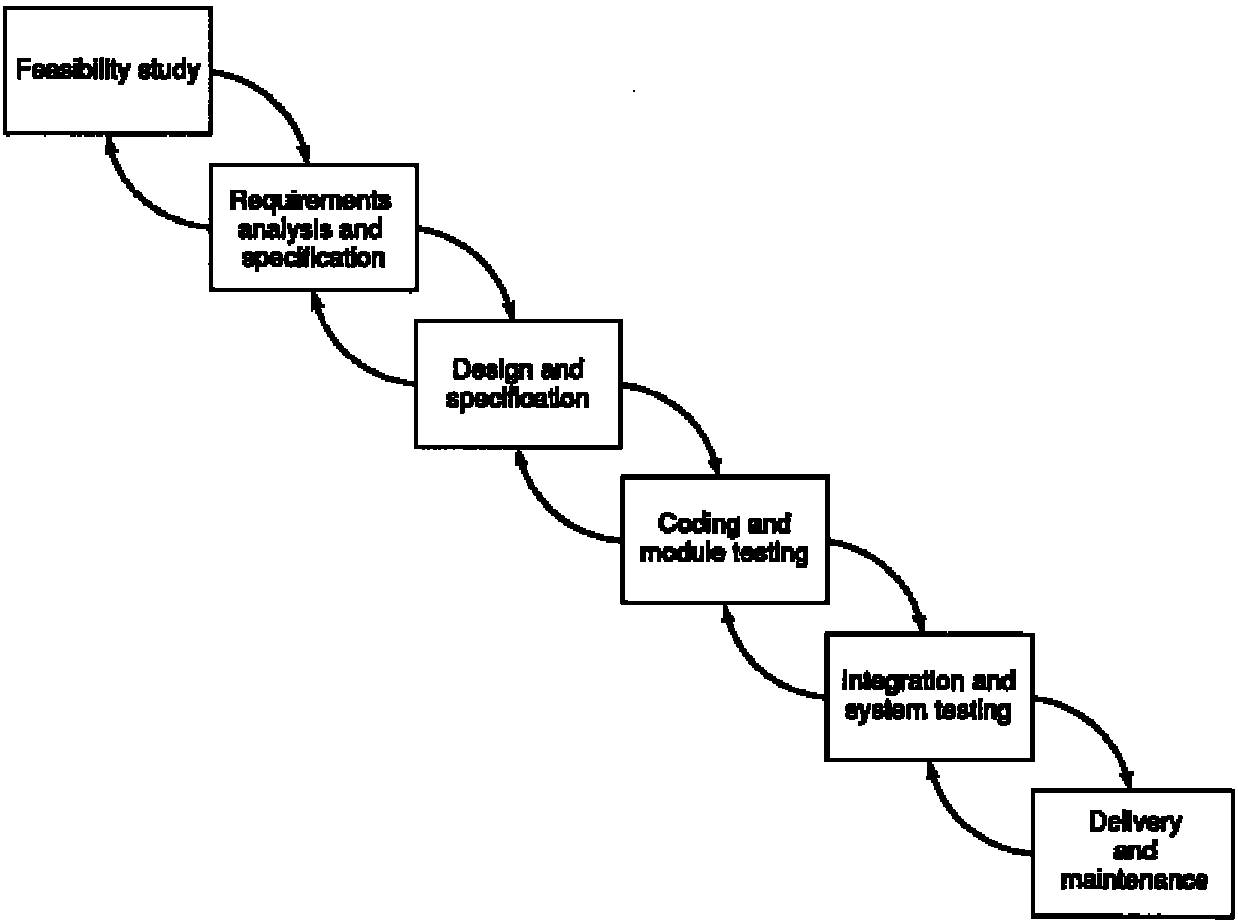

Figure 1. The Waterfall Model of Software Development (Royce 1970)

Stepwise Refinement

In this approach, software systems are developed through the progressive refinement and enhancement of high-level system specifications into source code components (Wirth 1971, Mili 1986). However, the choice and order of which steps to choose and which refinements to apply remain unstated. Instead, formalization is expected to emerge within the heuristics and skills that are acquired and applied through increasingly competent practice. This model has been most effective and widely applied in helping to teach individual programmers how to organize their software development work. Many interpretations of the classic software life cycle thus subsume this approach within their design and implementations.

Incremental Development and Release

Developing systems through incremental release requires first providing essential operating functions, then providing system users with improved and more capable versions of a system at regular intervals (Basili 1975). This model combines the classic software life cycle with iterative enhancement at the level of system development organization. It also supports a strategy to

periodically distribute software maintenance updates and services to dispersed user communities. This in turn accommodates the provision of standard software maintenance contracts. It is therefore a popular model of software evolution used by many commercial software firms and system vendors. This approach has also been extended through the use of software prototyping tools and techniques (described later), which more directly provide support for incremental development and iterative release for early and ongoing user feedback and evaluation (Graham 1989). Figure 2 provides an example view of an incremental development, build, and release model for engineering large Ada-based software systems, developed by Royce (1990) at TRW.

Elsewhere, the Cleanroom software development method at use in IBM and NASA laboratories provides incremental release of software functions and/or subsystems (developed through stepwise refinement) to separate in-house quality assurance teams that apply statistical measures and analyses as the basis for certifying high-quality software systems (Selby 1987, Mills 1987).

Industrial and Military Standards, and Capability Models

Industrial firms often adopt some variation of the classic model as the basis for standardizing their software development practices (Royce 1970, Boehm 1976, Distaso 1980, Humphrey 1985, Scacchi 1984, Somerville 1999). Such standardization is often motivated by needs to simplify or eliminate complications that emerge during large software development or project management.

From the 1970's through the present, many government contractors organized their software development activities according to succession of military software standards such as MIL-STD-2167A, MIL-STD 498, and IEEE-STD-016. ISO12207 (Moore 1997) is now the standard that most such contractors now follow. These standards are an outgrowth of the classic life cycle activities, together with the documents required by clients who procure either software systems or complex platforms with embedded software systems. Military software system are often constrained in ways not found in industrial or academic practice, including: (1) required use of military standard computing equipment (which is often technologically dated and possesses limited processing capabilities); (2) are embedded in larger systems (e.g., airplanes, submarines, missiles, command and control systems) which are mission-critical (i.e., those whose untimely failure could result in military disadvantage and/or life-threatening risks); (3) are developed under contract to private firms through cumbersome procurement and acquisition procedures that can be subject to public scrutiny and legislative intervention; and (4) many embedded software systems for the military are among the largest and most complex systems in the world (Moore 1997). Finally, the development of custom software systems using commercial off-the-shelf (COTS) components or products is a recent direction for government contractors, and thus represents new challenges for how to incorporate a component-based development into the overall software life cycle. Accordingly, new software life cycle models that exploit COTS components will continue to appear in the next few years.

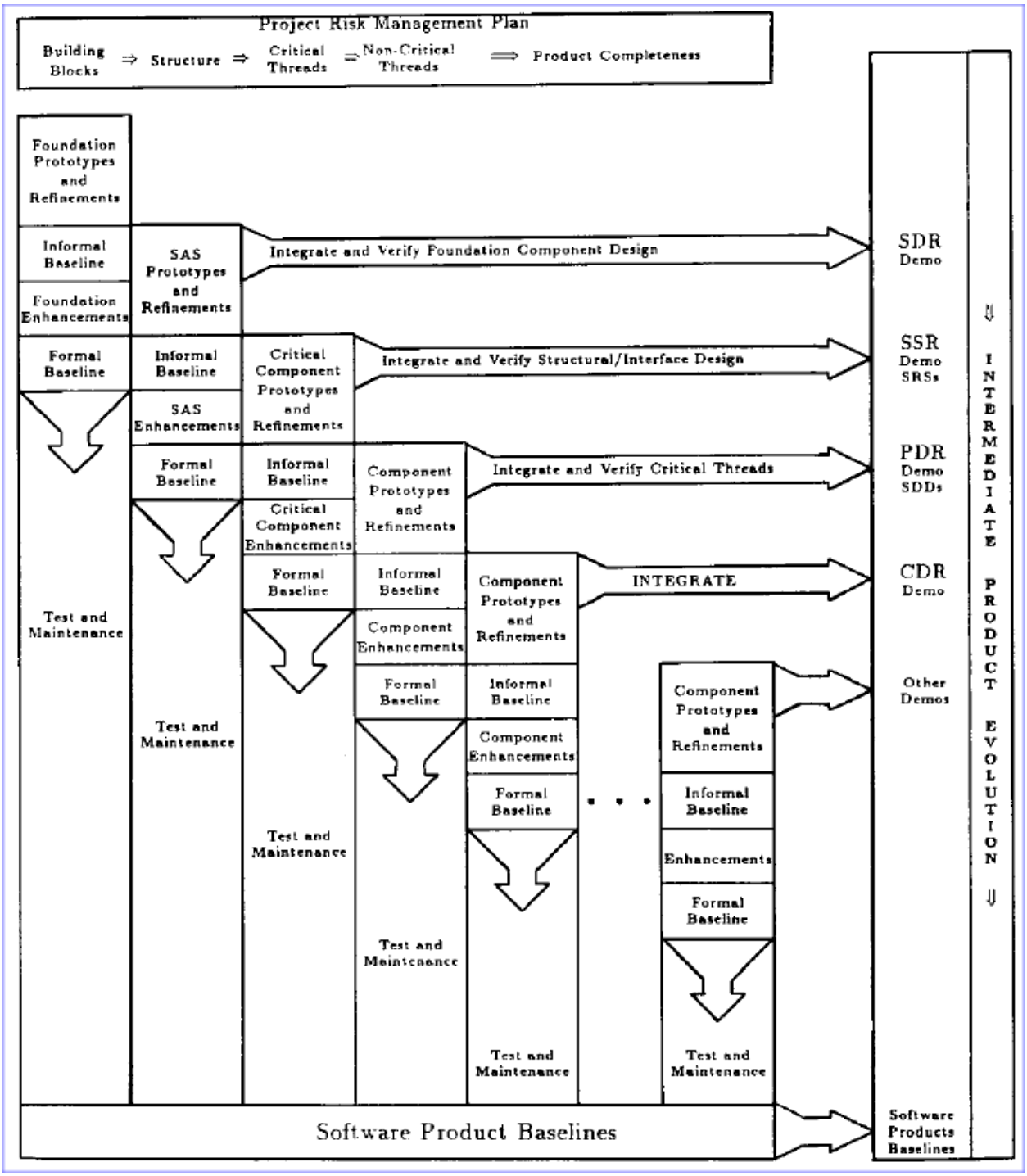

Figure 2. An Incremental Development, Build, and Release Model (Royce 1990)

In industrial settings, standard software development models represent often provide explicit detailed guidelines for how to deploy, install, customize or tune a new software system release in its operating application environment. In addition, these standards are intended to be compatible with provision of software quality assurance, configuration management, and independent verification and validation services in a multi-contractor development project. Early efforts in monitoring and measuring software process performance found in industrial practice appear in (Humphrey 1985, Radice 1985, Basili 1988). These efforts in turn help pave the way for what many software development organizations now practice, or have been certified to practice, software process capability assessments, following the Capability Maturity Model developed by the Software Engineering Institute (Paulk 1995) (see Capability Maturity Model for Software).

Alternatives to the Traditional Software Life Cycle Models

There are at least three alternative sets of models of software development. These models are alternatives to the traditional software life cycle models. These three sets focus of attention to either the products, production processes, or production settings associated with software development. Collectively, these alternative models are finer-grained, often detailed to the point of computational formalization, more often empirically grounded, and in some cases address the role of new automated technologies in facilitating software development. As these models are not in widespread practice, we examine each set of models in the following sections.

Software Product Development Models

Software products represent the information-intensive artifacts that are incrementally constructed and iteratively revised through a software development effort. Such efforts can be modeled using software product life cycle models. These product development models represent an evolutionary revision to the traditional software life cycle models (MacCormack 2001). The revisions arose due to the availability of new software development technologies such as software prototyping languages and environments, reusable software, application generators, and documentation support environments. Each of these technologies seeks to enable the creation of executable software implementations either earlier in the software development effort or more rapidly. Therefore in this regard, the models of software development may be implicit in the use of the technology, rather than explicitly articulated. This is possible because such models become increasingly intuitive to those developers whose favorable experiences with these technologies substantiate their use. Thus, detailed examination of these models is most appropriate when such technologies are available for use or experimentation.

Rapid Prototyping and Joint Application Development

Prototyping is a technique for providing a reduced functionality or a limited performance version of a software system early in its development (Balzer 1983, Budde 1984, Hekmatpour 1987). In contrast to the classic system life cycle, prototyping is an approach whereby more emphasis, activity, and processing are directed to the early stages of software development (requirements analysis and functional specification). In turn, prototyping can more directly accommodate early

user participation in determining, shaping, or evaluating emerging system functionality. Therefore, these up-front concentrations of effort, together with the use of prototyping technologies, seeks to trade-off or otherwise reduce downstream software design activities and iterations, as well as simplify the software implementation effort. (see Rapid Prototyping)

Software prototypes come in different forms including throwaway prototypes, mock-ups, demonstration systems, quick-and-dirty prototypes, and incremental evolutionary prototypes (Hekmatpour 1987). Increasing functionality and subsequent evolvability is what distinguishes the prototype forms on this list.

Prototyping technologies usually take some form of software functional specifications as their starting point or input, which in turn is simulated, analyzed, or directly executed. These technologies can allow developers to rapidly construct early or primitive versions of software systems that users can evaluate. User evaluations can then be incorporated as feedback to refine the emerging system specifications and designs. Further, depending on the prototyping technology, the complete working system can be developed through a continual revising/refining the input specifications. This has the advantage of always providing a working version of the emerging system, while redefining software design and testing activities to input specification refinement and execution. Alternatively, other prototyping approaches are best suited for developing throwaway or demonstration systems, or for building prototypes by reusing part/all of some existing software systems. Subsequently, it becomes clear why modern models of software development like the Spiral Model (described later) and the ISO 12207 expect that prototyping will be a common activity that facilitates the capture and refinement of software requirements, as well as overall software development.

Joint Application Development (JAD) is a technique for engaging a group or team of software developers, testers, customers, and prospective end-users in a collaborative requirements elicitation and prototyping effort (Wood and Silver 1995). JAD is quintessentially a technique for facilitating group interaction and collaboration. Consultants often employ JAD or external software system vendors who have been engaged to build a custom software system for use in a particular organizational setting. The JAD process is based on four ideas:

1.People who actually work at a job have the best understanding of that job.

2.People who are trained in software development have the best understanding of the possibilities of that technology.

3. Software-based information systems and business processes rarely exist in isolation -- they transcend the confines of any single system or office and effect work in related departments. People working in these related areas have valuable insight on the role of a system within a larger community.

4.The best information systems are designed when all of these groups work together on a project as equal partners.

Following these ideas, it should be possible for JAD to cover the complete development life cycle of a system. The JAD is usually a 3 to 6 month well-defined project, when systems can be

constructed from commercially available software products that do not require extensive coding or complex systems integration. For large-scale projects, it is recommended that the project be organized as an incremental development effort, and that separate JAD's be used for each increment (Wood and Silver 1995). Given this formulation, it is possible to view open source software development projects that rely on group email discussions among globally distributed users and developers, together with Internet-based synchronized version updates (Fogel 1999, Mockus 2000), as an informal variant of JAD.

Assembling Reusable Components

The basic approach of reusability is to configure and specialize pre-existing software components into viable application systems (Biggerstaff 1984, Neighbors 1984, Goguen 1986). Such source code components might already have associated specifications and designs associated with their implementations, as well as have been tested and certified. However, it is also clear that software domain models, system specifications, designs, test case suites, and other software abstractions may themselves be treated as reusable software development components. These components may have a greater potential for favorable impact on reuse and semi-automated system generation or composition (Batory et al., 1994, Neighbors 1984). Therefore, assembling reusable software components is a strategy for decreasing software development effort in ways that are compatible with the traditional life cycle models.

The basic dilemmas encountered with reusable software componentry include (a) acquiring, analyzing and modeling a software application domain, (b) how to define an appropriate software part naming or classification scheme, (c) collecting or building reusable software components, (d) configuring or composing components into a viable application, and (e) maintaining and searching a components library. In turn, each of these dilemmas is mitigated or resolved in practice through the selection of software component granularity.

The granularity of the components (i.e., size, complexity, and functional capability) varies greatly across different approaches. Most approaches attempt to utilize components similar to common (textbook) data structures with algorithms for their manipulation: small-grain components. However, the use/reuse of small-grain components in and of itself does not constitute a distinct approach to software development. Other approaches attempt to utilize components resembling functionally complete systems or subsystems (e.g., user interface management system): large-grain components. The use/reuse of large-grain components guided by an application domain analysis and subsequent mapping of attributed domain objects and operations onto interrelated components does appear to be an alternative approach to developing software systems (Neighbors 1984), and thus is an area of active research.

There are many ways to utilize reusable software components in evolving software systems. However, the cited studies suggest their initial use during architectural or component design specification as a way to speed implementation. They might also be used for prototyping purposes if a suitable software prototyping technology is available.

Application Generation

Application generation is an approach to software development similar to reuse of

parameterized, large-grain software source code components. Such components are configured and specialized to an application domain via a formalized specification language used as input to the application generator. Common examples provide standardized interfaces to database management system applications, and include generators for reports, graphics, user interfaces, and application-specific editors (Batory, et al. 1994, Horowitz 1985).

Application generators give rise to a model of software development whereby traditional software design activities are either all but eliminated, or reduced to a data base design problem. The software design activities are eliminated or reduced because the application generator embodies or provides a generic software design that should be compatible with the application domain. However, users of application generators are usually expected to provide input specifications and application maintenance services. These capabilities are possible since the generators can usually only produce software systems specific to a small number of similar application domains, and usually those that depend on a data base management system.

Software Documentation Support Environments

Much of the focus on developing software products draws attention to the tangible software artifacts that result. Most often, these products take the form of documents: commented source code listings, structured design diagrams, unit development folders, etc. These documents characterize what the developed system is suppose to do, how it does it, how it was developed, how it was put together and validated, and how to install, use, and maintain it. Thus, a collection of software documents records the passage of a developed software system through a set of life cycle stages.

It seems reasonable that there will be models of software development that focus attention to the systematic production, organization, and management of the software development documents. Further, as documents are tangible products, it is common practice when software systems are developed under contract to a private firm, that the delivery of these documents is a contractual stipulation, as well as the basis for receiving payment for development work already performed. Thus, the need to support and validate conformance of these documents to software development and quality assurance standards emerges. However, software development documents are often a primary medium for communication between developers, users, and maintainers that spans organizational space and time. Thus, each of these groups can benefit from automated mechanisms that allow them to browse, query, retrieve, and selectively print documents (Garg and Scacchi, 1989, 1990). As such, we should not be surprise to see construction and deployment of software environments that provide ever increasing automated support for engineering the software documentation life cycle (e.g., Penedo 1985, Horowitz 1986, Garg and Scacchi, 1989, 1990), or how these capabilities have since become part of the commonly available computer-aided software engineering (CASE) tools suites like Rational Rose, and others based on the use of the Unified Modeling Language (UML).

Rapid Iteration, Incremental Evolution, and Evolutionary Delivery

There are a growing number of technological, social and economic trends that are shaping how a new generation of software systems are being developed that exploit the Internet and World

Wide Web. These include the synchronize and stabilize techniques popularized by Microsoft and Netscape at the height of the fiercely competitive efforts to dominate the Web browser market of the mid 1990's (Cusumano and Yoffie, 1999, MacCormack 2001). They also include the development of open source software systems that rely on a decentralized community of volunteer software developers to collectively develop and test software systems that are incrementally enhanced, released, experienced, and debugged in an overall iterative and cyclic manner (DiBona 1999, Fogel 1999, Mockus 2000). The elapsed time of these incremental development life cycles on some projects may be measured in weeks, days, or hours! The centralized planning, management authority and coordination imposed by the traditional system life cycle model has been discarded in these efforts, replaced instead by a more organic, participatory, reputation-based, and community oriented engineering practice. Software engineering in the style of rapid iteration and incremental evolution is one that focuses on and celebrates the inevitability of constantly shifting system requirements, unanticipated situations of use and functional enhancement, and the need for developers to collaborate with one another, even when they have never met (Truex 1999). As such, we are likely to see more research and commercial development aimed at figuring out whether or how software process models can accommodate rapid iteration, incremental evolution, or synchronize and stabilize techniques whether applied to closed, centrally developed systems, or to open, de-centrally developed systems.

Program Evolution Models

In contrast to the preceding four prescriptive product development models, Lehman and Belady sought to develop a descriptive model of software product evolution. They conducted a series of empirical studies of the evolution of large software systems at IBM during the 1970's (Lehman1985). (see Software Evolution) Based on their investigations, they identify five properties that characterize the evolution of large software systems. These are:

1. *Continuing change*: a large software system undergoes continuing change or becomes progressively less useful

2. *Increasing complexity*: as a software system evolves, its complexity increases unless work is done to maintain or reduce it

3. *Fundamental law of program evolution*: program evolution, the programming process, and global measures of project and system attributes are statistically self-regulating with determinable trends and invariances

4. *Invariant work rate*: the rate of global activity in a large software project is statistically invariant

5. *Incremental growth limit*: during the active life of a large program, the volume of modifications made to successive releases is statistically invariant.

However, it is important to observe that these are global properties of large software systems, not causal mechanisms of software development. More recent advances in the study of program evolution can be found elsewhere in the article by Lehman and Ramil (See Software Evolution

chapter).

Software Production Process Models

There are two kinds of software production process models: non-operational and operational. Both are software process models. The difference between the two primarily stems from the fact that the operational models can be viewed as computational scripts or programs: programs that implement a particular regimen of software engineering and development. Non-operational models on the other hand denote conceptual approaches that have not yet been sufficiently articulated in a form suitable for codification or automated processing.

Non-Operational Process Models

There are two classes of non-operational software process models of the great interest. These are the spiral model and the continuous transformation models. There is also a wide selection of other non-operational models, which for brevity we label as miscellaneous models. Each is examined in turn.

The Spiral Model. The spiral model of software development and evolution represents a risk-driven approach to software process analysis and structuring (Boehm 1987, Boehm *et al*, 1998). This approach, developed by Barry Boehm, incorporates elements of specification-driven, prototype-driven process methods, together with the classic software life cycle. It does so by representing iterative development cycles as an expanding spiral, with inner cycles denoting early system analysis and prototyping, and outer cycles denoting the classic software life cycle. The radial dimension denotes cumulative development costs, and the angular dimension denotes progress made in accomplishing each development spiral. See Figure 3.

Risk analysis, which seeks to identify situations that might cause a development effort to fail or go over budget/schedule, occurs during each spiral cycle. In each cycle, it represents roughly the same amount of angular displacement, while the displaced sweep volume denotes increasing levels of effort required for risk analysis. System development in this model therefore spirals out only so far as needed according to the risk that must be managed. Alternatively, the spiral model indicates that the classic software life cycle model need only be followed when risks are greatest, and after early system prototyping as a way of reducing these risks, albeit at increased cost. The insights that the Spiral Model offered has in turned influenced the standard software life cycle process models, such as ISO12207 noted earlier. Finally, efforts are now in progress to integrate computer-based support for stakeholder negotiations and capture of trade-off rationales into an operational form of the WinWin Spiral Model (Boehm et al, 1998). (see Risk Management in Software Development)

Miscellaneous Process Models. Many variations of the non-operational life cycle and process models have been proposed, and appear in the proceedings of the international software process workshops sponsored by the ACM, IEEE, and Software Process Association. These include fully

interconnected life cycle models that accommodate transitions between any two phases subject to satisfaction of their pre- and post-conditions, as well as compound variations on the traditional life cycle and continuous transformation models. However, reports indicate that in general most software process models are exploratory, though there is now a growing base of experimental or industrial experience with these models (Basili 1988, Raffo *et al* 1999, Raffo and Scacchi 2000).

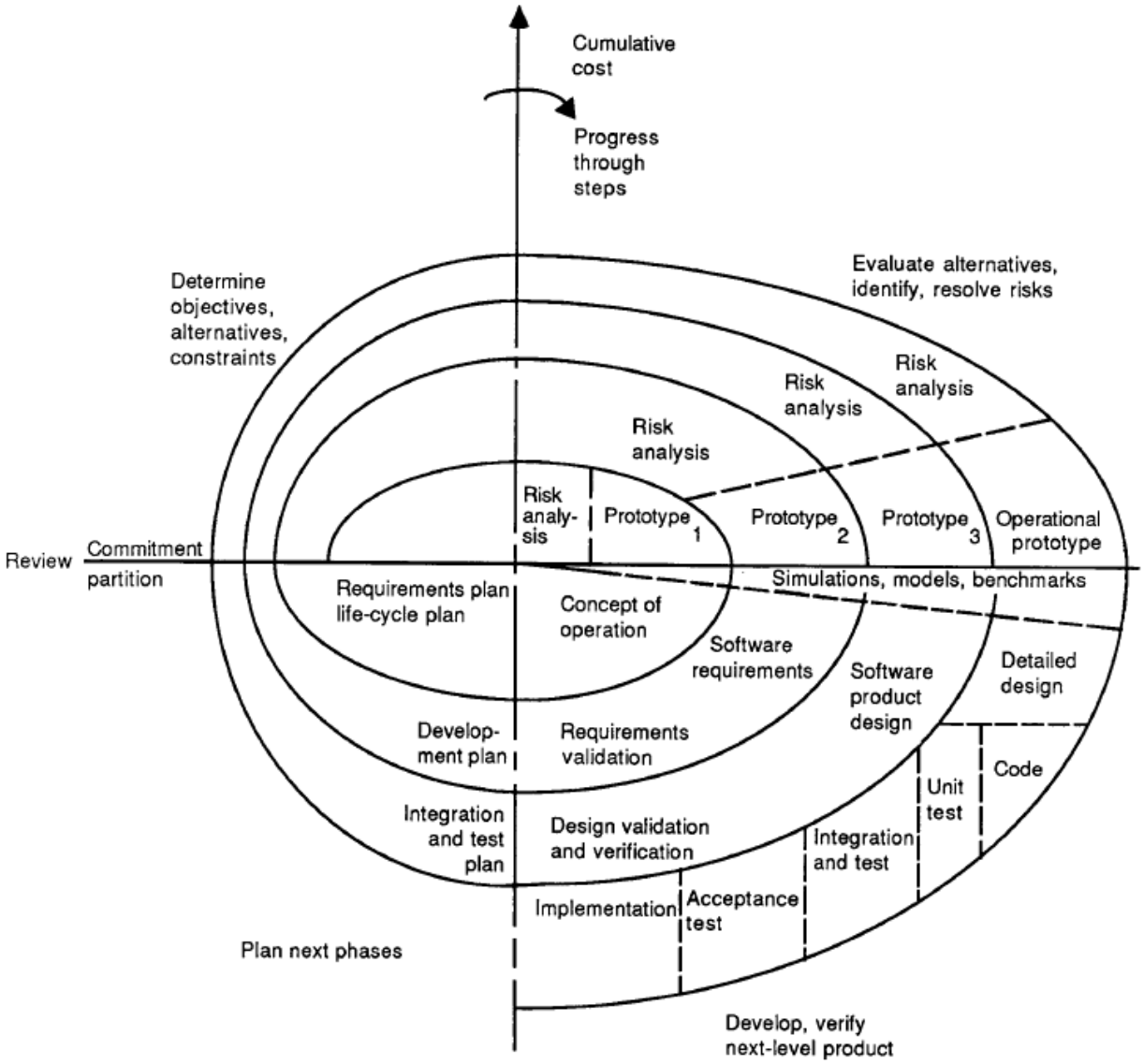

Figure 3.The Spiral Model diagram from (Boehm 1987)

Operational Process Models

In contrast to the preceding non-operational process models, many models are now beginning to appear which codify software engineering processes in computational terms--as programs or

executable models. Three classes of operational software process models can be identified and examined. Following this, we can also identify a number of emerging trends that exploit and extend the use of operational process models for software engineering.

Operational specifications for rapid prototyping. The operational approach to software development assumes the existence of a formal specification language and processing environment that supports the evolutionary development of specifications into an prototype implementation (Bauer 1976, Balzer 1983, Zave 1984). Specifications in the language are coded, and when computationally evaluated, constitute a functional prototype of the specified system. When such specifications can be developed and processed incrementally, the resulting system prototypes can be refined and evolved into functionally more complete systems. However, the emerging software systems are always operational in some form during their development. Variations within this approach represent either efforts where the prototype is the end sought, or where specified prototypes are kept operational but refined into a complete system.

The specification language determines the power underlying operational specification technology. Simply stated, if the specification language is a conventional programming language, then nothing new in the way of software development is realized. However, if the specification incorporates (or extends to) syntactic and semantic language constructs that are specific to the application domain, which usually are not part of conventional programming languages, then domain-specific rapid prototyping can be supported.

An interesting twist worthy of note is that it is generally within the capabilities of many operational specification languages to specify "systems" whose purpose is to serve as a model of an arbitrary abstract process, such as a software process model. In this way, using a prototyping language and environment, one might be able to specify an abstract model of some software engineering processes as a system that produces and consumes certain types of documents, as well as the classes of development transformations applied to them. Thus, in this regard, it may be possible to construct operational software process models that can be executed or simulated using software prototyping technology. Humphrey and Kellner describe one such application and give an example using the graphic-based state-machine notation provided in the STATECHARTS environment (Humphrey 1989).

Software automation. Automated software engineering (also called knowledge-based software engineering) attempts to take process automation to its limits by assuming that process specifications can be used directly to develop software systems, and to configure development environments to support the production tasks at hand. The common approach is to seek to automate some form of the continuous transformation model (Bauer 1976, Balzer 1985). In turn, this implies an automated environment capable of recording the formalized development of operational specifications, successively transforming and refining these specifications into an implemented system, assimilating maintenance requests by incorporating the new/enhanced specifications into the current development derivation, then replaying the revised development toward implementation (Balzer 1983b, Balzer 1985). However, current progress has been limited to demonstrating such mechanisms and specifications on software coding, maintenance, project communication and management tasks (Balzer 1983b, Balzer 1985, Sathi 1985, Mi 1990, Scacchi and Mi 1997), as well as to software component catalogs and formal models of software

development processes (Ould 1988, Wood 1988, Mi 1996). Last, recent research has shown how to combine different life cycle, product, and production process models within a process-driven framework that integrates both conventional and knowledge-based software engineering tools and environments (Garg 1994, Heineman 1994, Scacchi and Mi 1997).

Software process automation and programming. Process automation and programming are concerned with developing formal specifications of how a system or family of software systems should be developed. Such specifications therefore provide an account for the organization and description of various software production task chains, how they interrelate, when then can iterate, etc., as well as what software tools to use to support different tasks, and how these tools should be used (Hoffnagel 1985, Osterweil 1987). Focus then converges on characterizing the constructs incorporated into the language for specifying and programming software processes. Accordingly, discussion then turns to examine the appropriateness of language constructs for expressing rules for backward and forward-chaining, behavior, object type structures, process dynamism, constraints, goals, policies, modes of user interaction, plans, off-line activities, resource commitments, etc. across various levels of granularity (Garg and Scacchi 1989, Kaiser 1988, Mi and Scacchi 1992, Williams 1988, Yu and Mylopoulus 1994),. This in turn implies that conventional mechanisms such as operating system shell scripts (e.g., Makefiles on Unix) do not support the kinds of software process automation these constructs portend.

Lehman (1987) and Curtis and associates, (1987) provide provocative critiques of the potential and limitations of current proposals for software process automation and programming. Their criticisms, given our framework, essentially point out that many process programming proposals (as of 1987) were focused almost exclusively to those aspects of software engineering that were amenable to automation, such as tool sequence invocation. They point out how such proposals often fail to address how the production settings and products constrain and interact with how the software production process is defined and performed, as revealed in recent empirical software process studies (Bendifallah 1987, Curtis, et al., 1988, Bendifallah 1989, Grinter 1996).

Beyond these, the dominant trend during the 1990's associated with software process automation was the development of process-centered software engineering environments (Garg 1996). Dozens of research projects and some commercial developments were undertaken to develop, experiment with, and evaluate the potential opportunities and obstacles associated with software environments driven by operational software process models. Many alternative process model formalisms were tried including knowledge-based representations, rule-based schemes, and Petri-net schemes and variations. In the early 1990's, emphasis focused on the development of distributed client-server environments that generally relied on a centralized server. The server might then interpret a process model for how to schedule, coordinate, or reactively synchronize the software engineering activities of developers working with client-side tools (Garg et al 1994, Garg 1996, Heineman 1994, Scacchi and Mi 1997). To no surprise, by the late 1990's emphasis has shifted towards environment architectures that employed decentralized servers for process support, workflow automation, data storage, and tool services (Bolcer 1998, Grundy 1999, Scacchi and Noll 1997). Finally, there was also some effort to expand the scope of operational support bound to process models in terms that recognized their growing importance as a new kind of software (Osterweil 1987). Here we began to see the emergence of process engineering environments that support their own class of life cycle activities and support mechanisms (Garg

and Jazayeri 1996, Garg et al 1994, Heineman 1994, Scacchi and Mi 1997, Scacchi and Noll 1997).

Emerging Trends and New Directions

In addition to the ongoing interest, debate, and assessment of process-centered or process-driven software engineering environments that rely on process models to configure or control their operation (Ambriola 1999, Garg and Jazayeri 1996), there are a number of promising avenues for further research and development with software process models. These opportunities areas and sample direction for further exploration include:

- Software process simulation (Raffo et al, 1999, Raffo and Scacchi 2000) efforts which seek to determine or experimentally evaluate the performance of classic or operational process models using a sample of alternative parameter configurations or empirically derived process data (cf. Cook and Wolf 1998). Simulation of empirically derived models of software evolution or evolutionary processes also offer new avenues for exploration (Chatters, Lehman, *et al.*, 2000, Mockus 2000).

- Web-based software process models and process engineering environments (Bolcer 1998, Grundy 1998, Penedo 2000, Scacchi and Noll 1997) that seek to provide software development workspaces and project support capabilities that are tied to adaptive process models. (see Engineering Web Applications with Java)

- Software process and business process reengineering (Scacchi and Mi 1997, Scacchi and Noll 1997, Scacchi 2000) which focuses attention to opportunities that emerge when the tools, techniques, and concepts for each disciplined are combined to their relative advantage. This in turn is giving rise to new techniques for redesigning, situating, and optimizing software process models for specific organizational and system development settings (Scacchi and Noll 1997, Scacchi 2000). (see Business Reengineering in the Age of the Internet)

- Understanding, capturing, and operationalizing process models that characterize the practices and patterns of globally distributed software development associated with open source software (DiBona 1999, Fogel 1999, Mockus 2000), as well as other emerging software development processes, such as extreme programming (Beck 1999) and Web-based virtual software development enterprises or workspaces (Noll and Scacchi 1999,2001, Penedo 2000).

Conclusions

The central thesis of this chapter is that contemporary models of software development must account for software the interrelationships between software products and production processes, as well as for the roles played by tools, people and their workplaces. Modeling these patterns can utilize features of traditional software life cycle models, as well as those of automatable software process models. Nonetheless, we must also recognize that the death of the traditional system life cycle model may be at hand. New models for software development enabled by the Internet,

group facilitation and distant coordination within open source software communities, and shifting business imperatives in response to these conditions are giving rise to a new generation of software processes and process models. These new models provide a view of software development and evolution that is incremental, iterative, ongoing, interactive, and sensitive to social and organizational circumstances, while at the same time, increasingly amenable to automated support, facilitation, and collaboration over the distances of space and time.

References

Ambriola, V., R. Conradi and A. Fuggetta, Assessing process-centered software engineering environments, *ACM Trans. Softw. Eng. Methodol.* 6, 3, 283-328, 1997.

Balzer, R., Transformational Implementation: An Example, *IEEE Trans. Software Engineering*, 7, 1, 3-14,1981.

Balzer, R., A 15 Year Perspective on Automatic Programming, *IEEE Trans. Software Engineering*, 11,11,1257-1267, 1985.

Balzer, R., T. Cheatham, and C. Green, Software Technology in the 1990's: Using a New Paradigm, *Computer*,16,11, 39-46, 1983.

Basili, V.R. and H.D. Rombach, The TAME Project: Towards Improvement-Oriented Software Environments, *IEEE Trans. Soft. Engr.,* 14, 6, 759-773, 1988.

Basili, V. R., and A. J. Turner, Iterative Enhancement: A Practical Technique for Software Development, *IEEE Trans. Software Engineering*, 1,4, 390-396, 1975.

Batory, D., V. Singhal, J. Thomas, S. Dasari, B. Geraci, M. Sirkin, The GenVoca model of software-system generators, *IEEE Software*, 11(5), 89-94, September 1994.

Bauer, F. L., Programming as an Evolutionary Process, *Proc. 2nd. Intern. Conf. Software Engineering*, IEEE Computer Society, 223-234, January, 1976.

Beck, K. *Extreme Programming Explained*, Addison-Wesley, Palo Alto, CA, 1999.

Bendifallah, S., and W. Scacchi, Understanding Software Maintenance Work, *IEEE Trans. Software Engineering*, 13,3, 311-323, 1987.

Bendifallah, S. and W. Scacchi, Work Structures and Shifts: An Empirical Analysis of Software Specification Teamwork, *Proc. 11th. Intern. Conf. Software Engineering*, IEEE Computer Society, 260-270, 1989.

Biggerstaff, T., and A. Perlis (eds.), Special Issues on Software Reusability, *IEEE Trans. Software Engineering*, 10, ,5, 1984.

Boehm, B., Software Engineering, *IEEE Trans. Computer*, C-25,12,1226-1241, 1976.

Boehm, B. W., *Software Engineering Economics*, Prentice-Hall, Englewood Cliffs, N. J., 1981

Boehm, B., A Spiral Model of Software Development and Enhancement, *Computer*, 20(9), 61-72, 1987.

Boehm, B., A. Egyed, J. Kwan, D. Port, A. Shah, and R. Madachy, Using the WinWin Spiral Model: A Case Study, *Computer*, 31(7), 33-44, 1998.

Bolcer, G.A., R.N. Taylor, Advanced workflow management technologies, *Software Process--Improvement and Practice*, 4,3, 125-171, 1998.

Budde, R., K. Kuhlenkamp, L. Mathiassen, and H. Zullighoven, *Approaches to Prototyping*, Springer-Verlag, New York, 1984.

Chatters, B.W., M.M. Lehman, J.F. Ramil, and P. Werwick, Modeling a Software Evolution Process: A Long-Term Case Study, Software Process-Improvement and Practice, 5(2-3), 91-102, 2000.

Cook, J.E., and A.Wolf, Discovering models of software processes from event-based data, *ACM Trans. Softw. Eng. Methodol*. 7, 3 (Jul. 1998), 215 - 249

B. Curtis, H. Krasner, V. Shen, and N. Iscoe, On Building Software Process Models Under the Lamppost*, Proc. 9th. Intern. Conf. Software Engineering*, IEEE Computer Society, Monterey, CA, 96-103, 1987.

Curtis, B., H. Krasner, and N. Iscoe, A Field Study of the Software Design Process for Large Systems, *Communications ACM*, 31, 11, 1268-1287, November, 1988.

Cusumano, M. and D. Yoffie, Software Development on Internet Time, *Computer*, 32(10), 60-69, 1999.

Distaso, J., Software Management--A Survey of Practice in 1980, *Proceedings IEEE*, 68,9,1103-1119, 1980.

DiBona, C., S. Ockman and M. Stone, *Open Sources: Voices from the Open Source Revolution*, O'Reilly Press, Sebastopol, CA, 1999.

Fogel, K., *Open Source Development with CVS*, Coriolis Press, Scottsdale, AZ, 1999.

Garg, P.K. and M. Jazayeri (eds.), *Process-Centered Software Engineering Envir*onment, IEEE Computer Society, pp. 131-140, 1996.

Garg, P.K., P. Mi, T. Pham, W. Scacchi, and G. Thunquest, The SMART approach for software process engineering*, Proc. 16th. Intern. Conf. Software Engineering,* 341 - 350,1994.

Garg, P.K. and W. Scacchi, ISHYS: Design of an Intelligent Software Hypertext Environment, *IEEE Expert*, 4, 3, 52-63, 1989.

Garg, P.K. and W. Scacchi, A Hypertext System to Manage Software Life Cycle Documents, *IEEE Software*, 7, 2, 90-99, 1990.

Goguen, J., Reusing and Interconnecting Software Components, *Computer*, 19,2, 16-28, 1986.

Graham, D.R., Incremental Development: Review of Non-monolithic Life-Cycle Development Models, *Information and Software T*echnology, 31, 1, 7-20, January,1989.

Grundy, J.C.; Apperley, M.D.; Hosking, J.G.; Mugridge, W.B. A decentralized architecture for software process modeling and enactment, *IEEE Internet Computing* , Volume: 2 Issue: 5 , Sept.-Oct. 1998, 53 -62.

Grinter, R., Supporting Articulation Work Using Software Configuration Management*, J. Computer Supported Cooperative Work*,5, 447-465, 1996.

Heineman, G., J.E. Botsford, G. Caldiera, G.E. Kaiser, M.I. Kellner, and N.H. Madhavji., Emerging Technologies that Support a Software Process Life Cycle. *IBM Systems J*., 32(3):501-529, 1994.

Hekmatpour, S., Experience with Evolutionary Prototyping in a Large Software Project, *ACM Software Engineering Notes*, 12,1, 38-41 1987

Hoffnagel, G. F., and W. Beregi, Automating the Software Development Process, *,IBM Systems J.*,24 ,2 1985 ,102-120

Horowitz, E. and R. Williamson, SODOS: A Software Documentation Support Environment--Its Definition, *IEEE Trans. Software Engineering*, 12, 8, 1986.

Horowitz, E., A. Kemper, and B. Narasimhan, A Survey of Application Generators, *IEEE Software*, 2,1 ,40-54, 1985.

Hosier, W. A., Pitfalls and Safeguards in Real-Time Digital Systems with Emphasis on Programming, *IRE Trans. Engineering Management,* EM-8, June, 1961.

Humphrey, W. S., The IBM Large-Systems Software Development Process: Objectives and Direction, *,IBM Systems J*., 24,2, 76-78, 1985.

Humphrey, W.S. and M. Kellner, Software Process Modeling: Principles of Entity Process Models, *Proc. 11th. Intern. Conf. Software Engineering*, IEEE Computer Society, Pittsburgh, PA, 331-342, 1989.

Kaiser, G., P. Feiler, and S. Popovich, Intelligent Assistance for Software Development and Maintenance, *IEEE Software*, 5, 3, 1988.

Kling, R., and W. Scacchi, The Web of Computing: Computer Technology as Social Organization, *Advances in Computers*, 21, 1-90, Academic Press, New York, 1982.

Lehman, M. M., Process Models, Process Programming, Programming Support, *Proc. 9th. Intern. Conf. Software Engineering*, 14-16, IEEE Computer Society, 1987.

Lehman, M. M., and L. Belady, *Program Evolution: Processes of Software Change*, Academic Press, New York, 1985

MacCormack, A., Product-Development Practices that Work: How Internet Companies Build Software, *Sloan Management Review*, 75-84, Winter 2001.

Mi, P. and W. Scacchi, A Knowledge Base Environment for Modeling and Simulating Software Engineering Processes, *IEEE Trans. Knowledge and Data Engineering*, 2,3, 283-294, 1990.

Mi, P. and W. Scacchi, Process Integration for CASE Environments, *IEEE Software*, 9,2, March,45-53,1992.

Mi, P. and W. Scacchi., A Meta-Model for Formulating Knowledge-Based Models of Software Development. *Decision Support Systems*, 17(4):313-330, 1996.

Mili, A., J. Desharnais, and J.R. Gagne, Formal Models of Stepwise Refinement of Programs, *ACM Computing Surveys*, 18, 3, 231-276, 1986.

Mills, H.D., M. Dyer and R.C. Linger, Cleanroom Software Engineering, *IEEE Software*, 4, 5, 19-25, 1987.

Mockus, A., R.T. Fielding, and J. Herbsleb, A Case Study of Open Software Development: The Apache Server*, Proc. 22nd. International Conf. Software Engineering*, Limerick, IR, 263-272, 2000.

Moore, J.W., P.R. DeWeese, and D. Rilling, "U. S. Software Life Cycle Process Standards," *Crosstalk: The DoD Journal of Software Engineering*, 10:7, July 1997

Neighbors, J., The Draco Approach to Constructing Software from Reusable Components, *IEEE Trans. Software Engineering*, 10, 5, 564-574, 1984.

Noll, J. and W. Scacchi, Supporting Software Development in Virtual Enterprises, *Journal of Digital Information*, 1(4), February 1999.

Noll, J. and W. Scacchi, Specifying Process-Oriented Hypertext for Organizational Computing, *J. Network and Computer Applications*, 24(1):39-61, 2001.

Osterweil, L., Software Processes are Software Too, *Proc. 9th. Intern. Conf. Software Engineering*, 2-13, IEEE Computer Society, 1987.

Ould, M.A., and C. Roberts, Defining Formal Models of the Software Development Process, *Software Engineering Environments*, P. Brererton (ed.), Ellis Horwood, Chichester, England, 13-26, 1988.

Paulk, M.C., C.V. Weber, B. Curtis*, The Capability Maturity Model: Guidelines for Improving the Software Proces*s, Addison-Wesley, New York, 1995.

Penedo, M.H., An Active Web-based Virtual Room for Small Team Collaboration, *Software Process --Improvement and Practice,* 5,4,: 251-261, 2000.

Penedo, M.H. and E.D. Stuckle, PMDB--A Project Master Database for Software Engineering Environments, *Proc. 8th. Intern. Conf. Soft. Engr.,* IEEE Computer Society, Los Alamitos, CA,

150-157, 1985.

R. Radice, N.K. Roth, A.C. O'Hara and W.A. Ciarfella, A Programming Process Architecture. *IBM Systems Journal,* 24(2), 79-90, 1985.

Raffo, D. and W. Scacchi, Special Issue on Software Process Simulation and Modeling, *Software Process--Improvement and Practice*, 5(2-3), 87-209, 2000.

Raffo, D., W. Harrison, M.I. Kellner, R. Madachy, R. Martin, W. Scacchi, and P. Wernick, Special Issue on Software Process Simulation Modeling, *Journal of Systems and Software*, 46(2-3), 89-211, 1999.

Royce, W. W., Managing the Development of Large Software Systems, *Proc. 9th. Intern. Conf. Software Engineering,* ,IEEE Computer Society, 1987 ,328-338 Originally published in Proc. WESCON, 1970.

Royce, W., TRW's Ada Process Model for Incremental Development of Large Software Systems, *Proc. 12th. Intern. Conf. Software Engineering,* Nice, France, 2-11, IEEE Computer Society, 1990.

Sathi, A., M. S. Fox, and M. Greenberg, Representation of Activity Knowledge for Project Management, *IEEE Trans. Patt. Anal. and Mach. Intell.,* 7,5,531-552, 1985.

Scacchi, W., Managing Software Engineering Projects: A Social Analysis, *IEEE Trans. Software Engineering*, SE-10,1, 49-59, January, 1984.

Scacchi, W., Understanding Software Process Redesign using Modeling, Analysis and Simulation. *Software Process --Improvement and Practice* 5(2/3):183-195, 2000.

Scacchi, W. and P. Mi., Process Life Cycle Engineering: A Knowledge-Based Approach and Environment, *Intelligent Systems in Accounting, Finance, and Management*, 6(1):83-107, 1997.

Scacchi, W. and J. Noll, Process-Driven Intranets: Life Cycle Support for Process Reengineering, *IEEE Internet Computing*, 1(5):42-49, 1997.

Selby,R.W., V.R. Basili, and T. Baker, CLEANROOM Software Development: An Empirical Evaluation, *IEEE Trans. Soft. Engr*., 3, 9, 1027-1037, 1987.

Somerville, I. *Software Engineering* (7th. Edition), Addison-Wesley, Menlo Park, CA, 1999.

Truex, D., R. Baskerville, and H. Klein, Growing Systems in an Emergent Organization, *Communications ACM*, 42(8), 117-123, 1999.

Winograd, T. and F. Flores, *Understanding Computers and Cognition: A New Foundation for Design*, Ablex Publishers, Lexington, MA, 1986.

Williams, L., Software Process Modeling: A Behavioral Approach, *Proc. 10th. Intern. Conf. Software Engineering*, IEEE Computer Society, 174-200, 1988.

Wirth, N., Program Development by Stepwise Refinement, *Communications of the ACM*,14,4, 221-227, 1971.

Wood, J. and D. Silver, *Joint Application Development*, Wiley and Sons, Inc. New York, 1995.

Wood, M., and I. Sommerville, A Knowledge-Based Software Components Catalogue, *Software Engineering Environments*, Ellis Horwood, P. Brererton (ed.), Chichester, England, 116-131, 1988.

Yu, E.S.K. and J. Mylopoulos, Understanding "why" in software process modelling, analysis, and design, *Proc. 16th. Intern. Conf. Software Engineering*, 159 -168, 1994.

Zave, P., The Operational Versus the Conventional Approach to Software Development, *Communications of the ACM*, 27, 104-118, 1984.